

Preventing Coordinated Attacks Via Distributed Alert Exchange

Joaquin Garcia-Alfaro¹, Michael A. Jaeger², Gero Mühl², and Joan Borrell¹

¹Autonomous University of Barcelona,
Dept. of Information and Communications Engineering,
Edifici Q, 08193 Bellaterra, Spain
{jgarcia, jborrell}@deic.uab.es

²Berlin University of Technology,
Institute for Telecommunication Systems,
Communication and Operating Systems Group,
EN6, Einsteinufer 17, D-10587 Berlin, Germany
{michael.jaeger, g_muehl}@acm.org

Abstract. Attacks on information systems followed by intrusions may cause large revenue losses. The prevention of both is not always possible by just considering information from isolated sources of the network. A global view of the whole system is necessary to recognize and react to the different actions of such an attack. The design and deployment of a decentralized system targeted at detecting as well as reacting to information system attacks might benefit from the loose coupling realized by publish/subscribe middleware. In this paper, we present the advantages and convenience in using this communication paradigm for a general decentralized attack prevention framework. Furthermore, we present the design and implementation of our approach based on existing publish/subscribe middleware and evaluate our approach for GNU/Linux systems.

Keywords: Network Security, Attack Prevention System, Publish/Subscribe, Message Oriented Middleware, IDMEF

1 Introduction

When attackers gain access to a corporate network by compromising authorized users, computers, or applications, the network and its resources can become an active part of a globally distributed or coordinated attack. Such an attack might be a coordinated port scan or distributed denial of service attack against third party networks—or even against computers on the same network. Both, distributed and coordinated attacks, rely on the combination of actions performed by a malicious adversary to violate the security policy of a target computer system. To prevent these attacks, a global view of the system as a whole is necessary. Hence, different events and specific information must be gathered and combined from various sources to detect patterns of such a distributed attack.

This comprises, for example, information about suspicious connections, initiation of processes, and the creation of new files.

We already presented an attack prevention framework that is targeted at detecting as well as reacting to distributed and coordinated attack scenarios [8]. It relies on gathering and correlating information held by multiple sources. In this approach, we apply a decentralized scheme based on message passing to share alerts in a secure communication infrastructure. This way, we can detect and prevent those attacks performing detection and reaction processes based on the knowledge gained through alert correlation. In this paper, we focus on the communication infrastructure of the attack prevention framework. We discuss the design of its constituting elements and the format of the exchanged messages. We finally evaluate a first implementation of the infrastructure deployed on a GNU/Linux system. The main motivation of our work aims at fostering the collaboration between the different components of a protection framework composed by security components in order to achieve a more complete view of the system in whole. Once achieved, one can detect and react on the different actions of a coordinated or distributed attack.

The structure of this paper is the following. We start in Section 2 with a discussion of related work. Section 3 gives an introduction to the publish/subscribe communication model and elaborates the convenience in using this model for our problem domain. In Section 4, we briefly overview the Intrusion Detection Message Exchange Format (IDMEF), which is the format we use for the exchange of audit information in our system. We then present the cooperation scheme of the system components in Section 5 including a presentation of the current state of our implementation based on xmlBlaster, an open source publish/subscribe message oriented middleware [21]. Section 6 presents first experimental results on the performance obtained with a first deployment of our implementation. We close this paper with conclusions and a discussion of future work in Section 7.

2 Related Work

Traditional client/server solutions for security monitoring and protection of large-scale networks rely on the deployment of multiple sensors. These sensors locally collect audit data and forward it to a central server, where it is further analyzed. Early intrusion detection systems such as DIDS [22] and STAT [12] use this architecture and process the monitoring data in one central node. DIDS (Distributed Intrusion Detection System), for instance, is one of the first systems referred to in the literature that is using monitoring architecture [22]. The main components of DIDS are a central analyzer component called DIDS director, a set of host-based sensors installed on each monitored host within the protected network, and a set of network-based sensors installed on each broadcasting segment of the target system. The communication channels between the central analyzer and the distributed sensors are bidirectional. This way, the sensors can push their reports asynchronously to the central analyzer while the director is still able to actively request more details from the sensors.

The issue of sensor distribution is the focus of NetSTAT [26], an application of STAT (State Transition Analysis Technique) [12] to network-based detection. It is based on NSTAT [13] and comprises several extensions. Based on the attack scenarios and the network fact modeled as a hyper-graph, NetSTAT automatically chooses places to probe network activities and applies an analysis of state transitions. This way, it is able to decide what information is needed to collect within the protected network. Although NetSTAT collects network events in a distributed way, it analyzes them in a centralized fashion similarly to DIDS.

The main limitation of both DIDS and NetSTAT is that their exchange of audit data can quickly become a bottleneck due to saturation problems associated with their centralized analyzers. Their monitoring schemes are straightforward as they simply push the data to a central node and perform the computation there. Both approaches try to reduce the audit data sent over the network to the central analysis unit by filtering removing information of no interest from the audit stream and applying compression schemes afterwards. Unfortunately, an efficient data reduction scheme capable of forwarding only relevant data for arbitrary threat scenarios seems infeasible. Thus, those systems are not able to avoid unnecessary overhead which may lead to an overload on the central analyzer in case too many sensors are deployed. Furthermore, having only one single analyzer also induces issues with respect to availability: If the central analyzer crashes or becomes the victim of a denial of service (DoS) attack, the whole system is completely blinded.

Some approaches published later try to solve those disadvantages. GrIDS [24], EMERALD [19], and AAFID [23], for example, propose the use of layered structures, where data is locally pre-processed and filtered, and further analyzed by intermediate components in a hierarchical fashion. The computational and network load is distributed over multiple analyzers and managers as well as over different domains to analyze. The analyzers and managers of each domain perform their detection for just a small part of the whole network. They forward the processed information to the entity that is on the top of the hierarchy, i.e., a master node which finally analyzes all the reported incidents of the system.

On the one hand, GrIDS (Graph-based Intrusion Detection System for large networks) is an evolution of DIDS [22] and aims at large-scale distributed systems. It performs detection of distributed scans and worms by aggregating computer and network information into activity graphs [24]. In contrast to the centralized approach of DIDS, GrIDS allows the construction of activity graphs that only represent hosts and the network activity between them. Each node of the graph represents a single host or a group of nodes, and the edges represent network traffic between nodes. The audit data of GrIDS is collected by means of both host- and network-based sensors, and then forwarded to the graph manager, which further feeds the collected information into the graph. The whole system deploys several graphs and graph managers in a hierarchical fashion in order to increase the scalability of the whole system. Therefore, each manager controls just a subset of the whole graph. Unfortunately, only little details are provided regarding

the communication infrastructure for the exchange of information between components which makes it hard to further analyze this system.

Similar to GrIDS, EMERALD (Event Monitoring Enabling Responses to Anomalous Live Disturbances) extends the work of IDes (Intrusion Detection Expert System) [16] and NIDES (Next-Generation Intrusion Detection Expert System) [1] by implementing a recursive framework in which generic building blocks can be deployed in a hierarchical fashion [19]. It combines host- and network-based sensors as well as anomaly- and misuse-based analyzers. EMERALD focuses on the protection of large-scale enterprise networks that are divided into independent domains, each one of them with its own security policy. The authors claim to rely on a very efficient communication infrastructure for the exchange of information between the system components. Unfortunately, they also provide only few details regarding their implementation. Thus, a general statement regarding the performance of their infrastructure cannot be made.

The AAFID (Architecture for Intrusion Detection using Autonomous Agents) also presents a hierarchical approach to remove the limitations of centralized approaches and, particularly, to provide better resistance to denial of service attacks [23]. It consists of four main components called agents, filters, transceivers, and monitors organized in a tree structure, where child and parent components communicate with each other. The communication subsystem of AAFID exhibits a very simplistic design and does not seem to be resistant to a denial of service attack as intended. Although the set of agents may communicate with each other to agree upon a common suspicion level regarding every host, all relevant data is simply forwarded to monitors via transceivers and demands for human interaction in order to detect distributed intrusions.

Although hierarchical approaches mitigate some weaknesses inherent to centralized schemes, they still do not avoid bottlenecks, scalability problems, and fault tolerance issues due to vulnerabilities at the root level. The first reason for this lies in the massive amount of audit data forwarded to the higher level components which cannot be reduced significantly through pre-filtering within small network domains. The second reason is the centralized root domain component which may crash or become unavailable, rendering the whole system unusable this way. In order to solve these problems with both central and hierarchical data analysis, a decentralized scheme free of dedicated processing nodes is needed.

Some decentralized message passing designs try to remove the limitations and disadvantages of centralized and hierarchical approaches identified above. Their approach of distributing the detection process has some advantages compared to centralized and hierarchical approaches. Mainly, decentralized architectures have no single point of failure and bottlenecks can thus be avoided. Some message passing designs such as CSM [27] and Quicksand [14] try to eliminate the need for dedicated elements by introducing a peer-to-peer architecture. Instead of having a central monitoring station to which all data has to be forwarded, there are independent uniform working entities at each host performing similar basic operations. To detect coordinated and distributed attacks, the different entities collaborate on the detection activities and cooperate to perform a decentralized correlation algorithm.

These designs seem to be a promising technology to implement decentralized architectures for the detection of attacks. However, the presented systems still exhibit very simplistic designs and suffer from several practical limitations. For instance, in some of them, every node has to have complete knowledge of the system: All nodes have to be connected to each other which can make the matrix of the connections that are used for providing the alert exchanging service grow explosively and become very costly to control and maintain. Another important disadvantage present in this design is that the different entities always need to know where a received notification has to be forwarded to (similar to a queue manager). This way, when the number of possible destinations grows, the network view can become extremely complex limiting the scalability of this approach. Other designs are based on flooding which furthermore makes the system easier to maintain on the cost of scalability as the message complexity quickly grows with the number of nodes in the system.

Most of these limitations can be solved efficiently by using a distributed publish/subscribe middleware. The advantages of publish/subscribe communication for our problem domain over other communication paradigms is that it keeps the producers of messages decoupled from the consumers and that the communication is information-driven. This way, it is possible to avoid problems regarding the scalability and the management inherent to other designs by means of a network of publishers, brokers, and subscribers. A publisher in a publish/subscribe system does not need to have any knowledge about any of the entities that consume the published information since the communication is anonymous. Likewise, the subscribers do not need to know anything about the publishers. New services can simply be added without any impact on or interruption of the service to other users. In [10,9], we presented an infrastructure inspired by the decentralized architectures discussed with the focus on removing the discussed limitations. In the following sections, we present further details on our work.

3 Publish/Subscribe Model

The publish/subscribe communication model implies many-to-many communication and is often implemented asynchronously. It is well suited for distributed systems [6] and often used in situations where a message (often referred to as a *notification* in the literature) published by a single entity is sent to multiple receivers that expressed their interests previously. Publish/subscribe systems allow for efficient and comfortable information dissemination to receivers that may have individual interests in arbitrary subsets of the messages published. In contrast to multicast communication, clients have the possibility to describe the events they are interested in more flexible than with subscribing to a multicast group (e.g., based on the contents of the notification). Clients acting as *subscribers* can choose to subscribe and later unsubscribe to filters matching a set of messages as time goes by, while all subscribers are independent of each other. Clients that publish notifications are called *publishers*.

3.1 Publish/Subscribe Systems

A publish/subscribe system that implements the publish/subscribe model consists of clients and a *notification service*, the clients are connected to. The latter is responsible of forwarding notifications from publishers to all interested subscribers and consists of at least one *broker* in a centralized implementation. For scalability reasons, it is common to implement the notification service in a distributed fashion with a *broker overlay network* that consists of multiple brokers that cooperate to provide the notification service.

The notification service provides a distributed infrastructure for notification routing which includes the management of subscriptions and the dissemination of notifications in a possibly asynchronous way. Clients can publish notifications and subscribe to filters that are matched against the notifications forwarded through the broker network. If a broker receives a new notification it checks if there is a local client that has subscribed to a filter that matches this notification. If so, the message is delivered to this client. Additionally, the broker forwards the message to neighbor brokers according to the applied routing algorithm. We refer to [18] for more details on publish/subscribe systems.

An example of a basic centralized publish/subscribe system is shown in Figure 1(a). Here, five clients are connected to a single broker: three clients that are publishing notifications and two clients that are subscribed to a subset of the notifications published on the broker. Subscribers can choose to subscribe to the notifications available through the broker or cancel existing subscriptions as needed. The broker matches the notifications it received from the publishers to the subscriptions, ensuring this way that every publication is delivered to all interested subscribers.

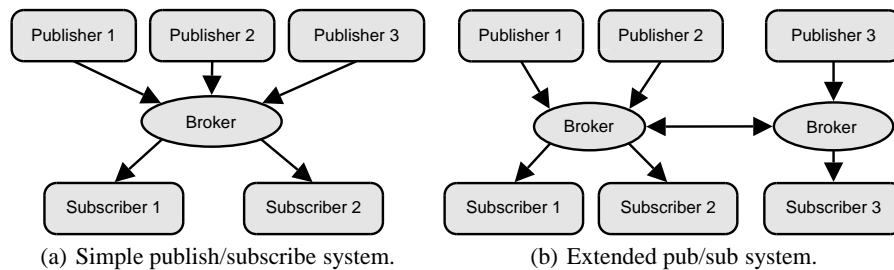


Fig. 1. Examples of simple publish/subscribe topologies.

This very basic publish/subscribe setup can be extended by connecting multiple brokers (cf. Figure 1(b)), enabling them to exchange messages. The extended design allows subscribers on one of the brokers to receive messages that have been published on another broker, further freeing the subscriber from the constraints of connecting to

the same broker the publisher is connected to. Most available implementations make this transparent for the programmer by keeping the same interface operations as in the centralized design. This way, an application can easily be distributed. In Figure 2, for instance, we show a distributed publish/subscribe topology, where a client p publishes a notification n that is matched by filter F client s subscribed to. The notification service then takes care of forwarding the notification properly over the links drawn solidly.

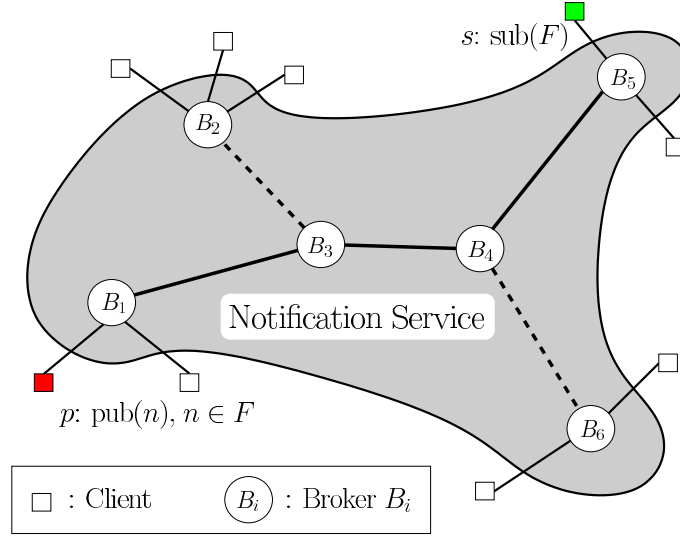


Fig. 2. Example of a distributed publish/subscribe topology.

Regarding the subscriptions, clients are able to formulate their interests based on the contents of the notifications or a special attribute they carry. *Topic-based publish/subscribe* systems represent the first variant of the publish/subscribe communication model. Here, publishers publish messages with respect to a topic and subscribers specify their interest in a topic and receive all messages published on this topic. Topic-based subscriptions are, in turn, easier to handle than content-based subscriptions. Since topics can be seen as groups in group communication [20], topic-based subscription may efficiently be built on top of a group communication mechanism such as, for example, IP multicast [5]. Thus, topics are equivalent to *channels*. An extension of the topic-based approach is *subject-based publish/subscribe*. Taking this approach, it is possible to arrange topics in a hierarchy (subject tree) such that subscriptions not only match notifications if the topics are the same, but also if the topic of the subscription is an ancestor of the notification topic in the subject tree. In this case, a subject becomes equivalent to a *theme*. In *type-based publish/subscribe*, notifications are equivalent to objects which allows for an easier integration in to object-oriented programming languages. Furthermore, it is possible with this approach to support multiple inheritance (depending on the programming language).

Content-based publish/subscribe systems allow filters to work on the content of notifications. This way, in content-based selection the structure of a subscription is not restricted to a topic or a subject—it can be any function over the content of a notification. A subscription can, thus, be formulated extremely fine-grained based on the content of notifications using a query language that can be arbitrarily complex. Moreover, there does not need to be a system-wide agreement on the set of topics as it is practically required for topic based routing.

Content-based subscriptions usually depend on the structure of the message. This can be binary data, name/value pairs, semi-structured data, or even programming language classes containing executable code. A subscription is often expressed in a subscription language that specifies a filter expression over messages.

For our work, we use content-based subscription over messages with semi-structured data. We propose the use of XML for the structure of a message as well as the application of XPath as the subscription language to specify filter expressions (cf. Section 5). In the following, we give an outlook on the main properties of the format built on top of the XML structure of our messages.

4 Representation of Messages

In order to exchange audit information in a standard manner, two main specifications have been considered in our job. The Common Intrusion Specification Language (CISL), on the one hand, which was initially proposed to allow the components of the Common Intrusion Detection Framework (CIDF) to exchange data in semantically well-defined ways [7]. The Intrusion Detection Message Exchange Format (IDMEF), on the other hand, was proposed by the IETF's Intrusion Detection Exchange Format Working Group (IDWG) to accomplish similar purposes [3].

Our approach is based on the IDMEF format for three main reasons. First, this format is the basis for the similarity operator used on the aggregation and fusion phases of our alert correlation approach presented in [8]. Second, there is a significant number of tools and implementations based on the IDMEF format, such as [17], which reduces the efforts of integrating it into our work. Third, the exchange of messages between the components of our framework is compliant with the intrusion detection framework proposed by the IDWG. Besides that, IDMEF allows the specification of messages generated by different network security components, such as *firewalls* and *network intrusion detection systems* (NIDSs), and it can be extended to incorporate additional data information, such as diagnoses and counter-measures, inside their proposed format.

Up to now, IDMEF is an *internet draft* approved by the IESG (Internet Steering Group) as an IETF RFC (Request For Comments). It is represented in an object-oriented fashion. The class hierarchy of IDMEF has been represented by using the Extensible Markup Language (XML). The rationale for choosing XML is explained in [3], as well as some examples of using IDMEF to describe IDS's alerts and the IDMEF's associate Docu-

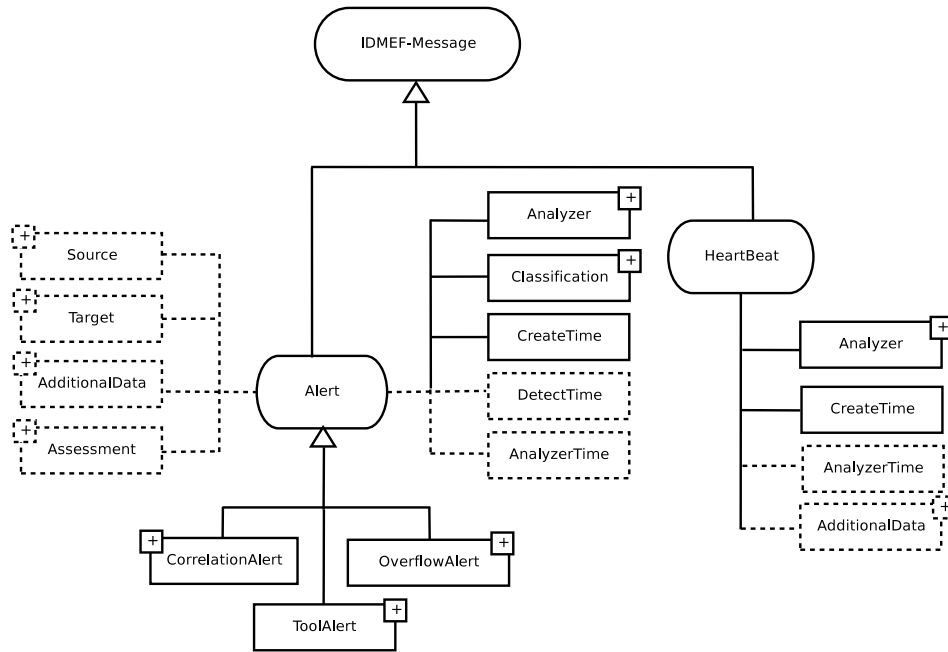


Fig. 3. The IDMEF's message class.

ment Type Definition (DTD)—although one may still find the current version of IDMEF defined by using DTDs, the authors also offer a new definition that uses XML Schemas instead of DTDs.

In Figure 3, we show the two main types of messages supported by IDMEF: *heartbeats* and *alerts*. Heartbeats, on the one hand, are periodic messages between components, in order to inform each other that they are operational. Alerts, on the other hand, carry audit information, such as the component that produced it, the classification of the detected activity, the source and target ports related to this activity, and other optional data. In the following, we discuss the main properties of IDMEF's alert class, regarding aspects relevant to our work such as determining the component which created the message, the time in which the message was created, and the kind of activity the message is pointing out.

We start by giving an overview of the analyzer class which identifies the component from which the message originates. Only one component is encoded for each message, i.e., the one from which the message originated. The class is composed, in turn, of three aggregate classes: *node*, which includes information about the node on which the component resides; *process*, which holds information about the process in which the component is executing; and *analyzer*, which carries information about other components which, in turn, forwarded the original information.

The rationale behind the recursive aggregation of component's references within the IDMEF's analyzer class is that when a component receives an IDMEF alert and wants to forward it to another component, it needs to substitute the original component information with its own since, as we pointed out above, just one component is encoded for each message. This way, and in order to preserve the original component information, it may be included in the new component definition as a reference to the previous component. This mechanism will allow component path tracking.

The class analyzer has eight attributes: *analyzerid*, *name*, *manufacturer*, *model*, *version*, *class*, *ostype*, and *osversion*. The *manufacturer*, *model*, *version*, and *class* attributes' contents are vendor-specific, but may be used together to identify different types of components. The *ostype* and *osversion* attributes' contents are, respectively, the operating system name and the operating system version in which the component's process is executed. Finally, the *analyzerid* and *name* attributes' contents provide, respectively, the unique identifier and the explicit name for the component in the system.

Regarding the timestamps of a message, the IDMEF standard defines the following three different classes to represent time: (1) *CreateTime*, which is the time when the message is created by a component; (2) *DetectTime*, which is the time when the event or events that caused the creation of a message were detected; (3) *AnalyzerTime*, which is the time when the original component forwarded this message. The final object for each instance contains information such as the number of seconds since the *epoch*, the local GMT offset, and the number of microseconds. Even though all the three timestamps can be provided by each component when generating a message, just the one defined by the *CreateTime* class is considered mandatory by the IDMEF standard.

The classes source and target contain, respectively, information about the possible origin and destination of the events that motivated the generation of the message. An event may have more than one source (e.g., a distributed denial of service attack), more than one target (e.g., a port sweep). Both, source and target classes, are composed of information about the *node*, the *user*, the *process*, and the *network service* that motivated the message. The target class includes, moreover, a list of affected *files*. Referring to their attributes, both source and target classes have the following two common attributes: (1) *ident*, which is a unique identifier for either the source or target class; (2) *interface*, which may be used by a component multiple interfaces to indicate which interface this source or target was seen on. Furthermore, the class source includes the attribute *spoofed*, which indicates whether the source is, as far as the component can determine, a spoofed address. Similarly, the class target includes the attribute *decoy*, to indicate whether the target is, as far as the analyzer can determine, a decoy.

The classification class contains the *name* of the event that motivated the creation of a message, or other information which allows the components to determine what the message is pointing out. It is composed of one aggregate class, the class *reference*, which contains information about external documentation sites, that will provide background information about such an event. Similarly, the assessment class is used to provide the component's assessment of an event, and it is composed of information about the *im-*

pact, *actions* that may be taken in response, and a measurement of the *confidence* the component has in its evaluation of the event.

Finally, the IDMEF's alert class can be augmented with additional information by means of the aggregate classes *AdditionalData*, *CorrelationAlert*, *ToolAlert*, and *OverflowAlert*. The information aggregated by those classes is often useful in order to associate different messages pointing out to similar activities—and reported by different components—as well as to extend the standard IDMEF model with additional features, such as complex data types and relationships. The *AdditionalData* class, first, includes information that does not fit into the IDMEF's data model. This may be an atomic piece of data, or a large amount of data. The *CorrelationAlert* class, on the second hand, may include additional information related to the correlation process in which this message is involved. The *OverflowAlert* and *ToolAlert* classes, on the third hand, include, respectively, information related to buffer overflow attacks, and information related to the use of attack tools or other malevolent programs (e.g., *trojan horses*, *rootkits*, and so on).

5 Communication Infrastructure

In this section we give an outlook to the operational details of the communication infrastructure presented in [8,10]. As our motivation is not targeted at developing a new publish/subscribe system, we try to reuse as much available code and tools as possible. For our experiments (cf. Section 6) we used *xmlBlaster*, an open source publish/subscribe message oriented middleware [21]. It connects a set of nodes that build up the infrastructure for exchanging alerts using the interface operations offered by the underlying middleware. Each *xmlBlaster* message consists of a header filtering that can be applied to, a body, and a system control section. The body of an *xmlBlaster* message is formulated using IDMEF format (cf. Section 4). Filters are XPath expressions that are evaluated over the message header to decide if a message has to be delivered to a subscriber. We discuss the essential interface operations offered by *xmlBlaster* in the following section.

5.1 Interface Operations

Conceptually, the alert communication infrastructure offered through *xmlBlaster* can be viewed as a black box with an *interface* (cf. Figure 4). It offers a number of *operations*, each of which may take a number of *parameters*. Clients can invoke *input operations* from the outside, and the system itself invokes *output operations* to deliver information to clients. To publish alerts, clients invoke the *pub(a)* operation, giving the alert *a* as parameter. The published alert can potentially be delivered to all clients connected to the system via an output operation called *notify(a)*. Clients register their interest in specific kinds of alerts by issuing subscriptions via the *sub(F)* operation, which takes a filter *F* as parameter. Each client can have multiple active subscriptions which must be revoked separately by using the *unsub(F)* operation.

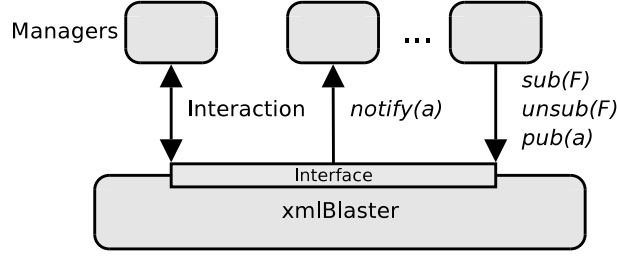


Fig. 4. Black box view of a publish/subscribe system.

All these operations are instantaneous and take parameters from the set of all clients \mathcal{C} , set of all alerts \mathcal{A} , and the set of all filters \mathcal{F} . Formally, a filter $F \in \mathcal{F}$ is a mapping defined by

$$F : a \longrightarrow \{\text{true}, \text{false}\} \quad \forall a \in \mathcal{A} \quad (1)$$

We say that a *notification* n *matches filter* $F \in \mathcal{F}$ iff $F(a) = \text{true}$. We require that each alert can only be published once and that every filter is associated with a unique identifier in order to enable the alert communication infrastructure to identify a specific subscription.

5.2 Components and Interactions

As shown in Figure 5, and according to the general framework introduced in [8], each node of the architecture is made up of a set of *local analyzers* (with their respective detection units or sensors), a set of *alert managers* (to perform alert processing and manipulation functions), and a set of *local reaction units* (or effectors). These components, the interactions between them, and the alert communication infrastructure, are described in the following.

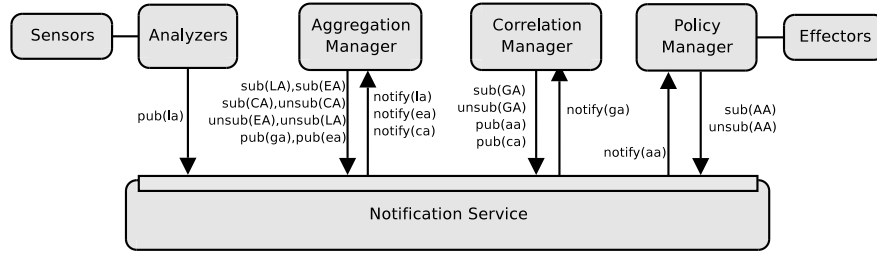


Fig. 5. Overview of the main components and their interactions.

Analyzers are local elements which are responsible for processing local audit data. They process the information gathered by associated sensors to infer possible alerts.

Their task is to identify occurrences which are relevant for the execution of the different steps of an attack and pass this information to the correlation manager via the publish/subscribe system. They are interested in *local alerts* which are detected in a sensor's input stream and published through the publish/subscribe system by invoking the *pub(la)* operation, giving the local alert *la* as parameter. Local alerts are exchanged using IDMEF messages (cf. Section 4).

Each notification *la* has a unique classification and a list of attributes with their respective types to identify the analyzer that originated the alert (*AnalyzerID*), the time the alert was created (*CreateTime*), the time the event(s) leading up to the alert was detected in the sensor's input stream (*DetectTime*), the current time on the analyzer (*AnalyzerTime*), and the source(s) and target(s) of the event(s) (*Source* and *Target*). All possible classifications and their respective attributes must be known by all system components, i.e., sensors, analyzers, and managers, and all analyzers are capable of publishing instances of local alerts of arbitrary types.

Managers are the components in charge of performing aggregation and correlation of local alerts and external events. As pointed out in [8], the use of multiple analyzers and sensors together with heterogeneous detection techniques increases the detection rate, but it also increases the number of information to process. In order to reduce the number of false negatives and distribute the load that is imposed by the alerts, our architecture provides a set of aggregation and correlation *managers*, which perform aggregation and correlation of both, local alerts (i.e., messages provided by the node's analyzers) and external messages (i.e., the information received from other collaborating nodes). In the following, we describe the basic interactions of the two main managers: *aggregation* and *correlation* managers.

Aggregation Manager. The basic functionality of each aggregation manager is to cluster alerts that correspond to the same occurrence of an action [8]. Each aggregation manager registers its interest in a subset \mathcal{L}_A of local alerts published by analyzers on the same node by invoking the *sub(LA)* operation, which takes the filter *LA* as parameter, with

$$LA(a) = \begin{cases} \text{true} & , a \in \mathcal{L}_A \\ \text{false} & , \text{otherwise.} \end{cases} \quad (2)$$

Similarly, the aggregation manager also registers its interest in a set of related external alerts \mathcal{E}_A by invoking the *sub(EA)* operation with filter *EA* as parameter, and

$$EA(a) = \begin{cases} \text{true} & , a \in \mathcal{E}_A \\ \text{false} & , \text{otherwise.} \end{cases} \quad (3)$$

Finally, it registers its interest in local correlated alerts \mathcal{C}_A by invoking the *sub(CA)* operation with

$$CA(a) = \begin{cases} \text{true} & , a \in \mathcal{C}_A \\ \text{false} & , \text{otherwise.} \end{cases} \quad (4)$$

Once subscribed to these three filters, the communication infrastructure will notify the subscribed managers of all matching alerts via the output operations *notify(la)*, *notify(ea)* and *notify(ca)* with $la \in \mathcal{L}_A$, $ea \in \mathcal{E}_A$ and $ca \in \mathcal{C}_A$. All notified alerts are processed and, depending on the clustering and synchronization mechanism, the aggregation manager can publish global and external alerts by invoking *pub(ga)* and *pub(ea)*. Finally, it can revoke active subscriptions separately by using the operations *unsub(CA)*, *unsub(EA)* and *unsub(LA)*.

Correlation Manager. The main task of this manager is the correlation of alerts described in [8,2]. It operates on the set of global alerts \mathcal{G}_A published by the aggregation manager. To register its interest in these alerts, it invokes *sub(GA)*, which takes the filter GA as parameter with

$$GA(a) = \begin{cases} \text{true} & , a \in \mathcal{G}_A \\ \text{false} & , \text{otherwise.} \end{cases} \quad (5)$$

The notification service will then notify the correlation manager of all matched alerts with the output operation *notify(ga)*, $ga \in \mathcal{G}_A$. Each time a new alert is received, the correlation mechanism finds a set of action models that can be correlated in order to form a scenario leading to an objective. It then includes this information into the *CorrelationAlert* field of a new IDMEF message and publishes the correlated alert by invoking *pub(ca)*, giving the notification $ca \in \mathcal{C}_A$ as parameter. To revoke the subscription, it uses *unsub(GA)*.

The correlation manager is also responsible for reacting on detected security violations. The algorithm used is based on the anti-correlation of actions to select appropriate counter-measures in order to reconfigure, for instance, the security policy [4]. As soon as a scenario is identified, the correlation mechanism may look for possible action models that can be anti-correlated with the individual actions of the supposed scenario, or even with the goal objective.

The set of anti-correlated actions represents the set of counter-measures available for the observed scenario. The definition of each anti-correlated action contains a description of the counter-measures which should be invoked (e.g., hardening the security policy). Such counter-measures are included into the *Assessment* field of a new IDMEF message and published by invoking *pub(aa)*, using the *assessment alert aa* as parameter.

Finally, a *policy manager* will register and revoke its interest in these assessment alerts by invoking *sub(AA)* and *unsub(AA)*. Once notified, the policy manager may perform the post-processing of the received alerts before sending them, for example, to a set of associated policy reconfiguration effectors.

6 Deployment and Evaluation

In order to evaluate the performance of our proposal, we deployed a set of analyzers and managers publishing and receiving IDMEF messages based on the *DARPA Intrusion Detection Evaluation Data Sets* [15]. This evaluation data set contains more than 300 instances of 38 different automated attacks that were launched against victim hosts in seven weeks of training data and two weeks of test data.

The complete set of messages were published as local and external alerts through the notification service of xmlBlaster, and then processed and republished in turn to the set of subscribed managers. The exchange of alerts proved to be satisfactory, obtaining a throughput performance higher than 150 messages per second on an Intel-Pentium M 1.4 GHz processor with 512 MB RAM, analyzers and managers on the same machine running Linux 2.6.8, using Java HotSpot Client VM 1.4.2 for the Java-based broker. Message delivery did not become a bottleneck as all messages were processed in time and the saturation point has never been reached.

The implementation of both, publishers and subscribers, was based on the *libidmef* C library [17] in order to build and parse compliant IDMEF messages. In turn, *libidmef* is built over the libxml library [25]. The libxml library provides two interfaces to parse XML data: a DOM style tree interface, and a SAX style event-based interface for our implementation. Up to now, we are using the DOM interface due to its easiness of use. Its main drawback is, however, that its memory usage is proportional to the size of the XML data. For this reason, we are currently rewriting our implementation to use the SAX-based interface. This would help us to decrease the amount of memory that is currently necessary to maintain the entire XML tree in memory.

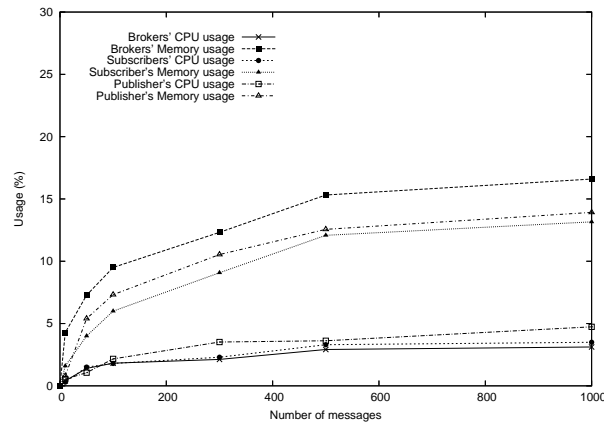


Fig. 6. Processing and memory consumption.

The communication between analyzers and managers through xmlBlaster brokers was based on the xmlBlaster internal socket protocol and implemented using the C socket library [21] for xmlBlaster, which provides asynchronous callbacks to Java-based brokers. The managers formulated their subscriptions using XPath expressions, filtering the messages they wished to receive from the broker.

In Figure 6, we show the processing time and memory space used by xmlBlaster brokers during the exchange of alerts. The first curve represents the percentage of CPU load used by each broker. The second curve represents the quantity of memory used by each broker. As we can notice in the first curve, the percentage of processing time used by the brokers is quite stable and negligible for the normal performance of a normal system. The second curve reflects, however, that the cost in memory is quite high. We consider that this consumption is due to the message management and we hope that the new version of our prototype based on a more efficient XML parsing and building scheme will lower it as discussed above.

7 Conclusions

We presented a message passing design for the exchange of audit information between the security components of a platform for the detection of and reaction on coordinated attacks. The design is based on a publish/subscribe model. Instead of having a central or master monitoring station to which all data has to be forwarded, there are independent uniform working entities at each host performing similar basic operations. The information gathered by each entity is disseminated to other interested entities through a notification service based on a publish/subscribe broker network which allows messages to be sent via a push or pull data exchange. The main advantage of this model for the exchange of audit information between components is, on the one hand, that it keeps the producer of messages separated from the consumers and, on the other hand, that the communication is information-driven. This way, it allows us to avoid problems regarding the scalability and the management inherent to other designs, by means of a network of publishers, brokers, and subscribers. A publisher in a publish/subscribe system does not need to have any knowledge about any of the entities that consume the published information. Likewise, the subscribers do not need to know anything about the publishers. Services can be added without any impact on or interruption of the service to other elements.

In Section 4, we discussed the main properties of the Intrusion Detection Message Exchange Format (IDMEF) as the format that is built on top of the XML structure of the messages exchanged between the components of our platform; we presented in Section 5 the operational details (interface operations and interaction) of our communication infrastructure; and we discussed the initial results of a first prototype of our approach in Section 6. We think that these results give us good hope that the use of a publish/subscribe system for the communication infrastructure indeed increases the scalability of the proposed architecture. Motivated by the high memory usage, and as

pointed out in Section 6, we are actually moving our current implementation to the SAX interface, since it does not maintain the entire XML tree in memory, which means that the load will considerably decrease.

As an extension of the work presented in this paper, we may first consider to secure the communication partners by utilizing the SSL protocol. This way, each node will receive a private and a public key. The public key of each node will be signed by a certification authority (CA) that is responsible for the protected network. Hence, the public key of the CA has to be distributed to every node as well. The secure SSL channel will allow the communicating peers to communicate privately and to authenticate each other, thus preventing malicious nodes from impersonating legal ones. The implications coming up with this new feature, such as compromised key management or certificate revocation, would be part of this future work. We may also consider as further work a more in-depth study about privacy mechanisms by exchanging alerts in a pseudonymous manner. By doing this, one may provide the destination and origin information of alerts (*Source* and *Target* field of IDMEF messages) without violating the privacy of publishers and subscribers located on different domains. Our study may cover the design of a pseudonymous identification scheme, trying to find a balance between identification and privacy. This also represents further work that remains to be done.

Acknowledgments

The collaboration between J. Garcia-Alfaro, F. Cuppens, and F. Autrel sharpened many of the arguments presented in this paper. The authors graciously acknowledge the financial support received from the following organizations: Spanish Ministry of Science and Education, and the Catalan Government's Agency for Management of University and Research Grants (AGAUR).

References

1. D. Anderson, T. Frivold, A. Valdes. *Next-generation Intrusion Detection Expert System (NIDES): a summary*. SRI International, Computer Science Laboratory, 1995.
2. F. Cuppens, F. Autrel, Y. Bouzida, J. Garcia-Alfaro, S. Gombault, and T. Sans. Anti-correlation as a criterion to select appropriate counter-measures in an intrusion detection framework. *Annals of Telecommunications*, 61(1-2):192–217, 2006.
3. H. Debar, D. Curry, and B. Feinstein. Intrusion detection message exchange format data model and extensible markup language. *Request for Comments 4765*, March 2007.
4. H. Debar, Y. Thomas, F. Cuppens, and N. Cuppens-Boulahia. Enabling Automated Threat Response through the Use of a Dynamic Security Policy. *Journal in Computer Virology (JCV)*, 3(3):195–210, August 2007.
5. S. Deering. Host Extensions for IP Multicasting. STD 5, RFC 1112, Stanford University, May 1988.
6. P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

7. R. Feiertag, C. Kahn, P. Porras, D. Schnackenberg, S. Staniford-Chen, and B. Tung. A Common Intrusion Specification Language. CIDF working group document, 1999.
8. J. Garcia-Alfaro, F. Autrel, J. Borrell, S. Castillo, F. Cuppens, and G. Navarro. Decentralized publish/subscribe system to prevent coordinated attacks via alert correlation. In *Sixth International Conference on Information and Communications Security*, volume 3269 of *LNCS*, pages 223–235, Málaga, Spain, October 2004. Springer-Verlag.
9. J. Garcia-Alfaro, J. Borrell, M. A. Jaeger, and G. Mühl. An alert communication infrastructure for a decentralized attack prevention framework. In *IEEE International Carnahan Conference on Security Technology*, pages 234–237, Las Palmas de G.C., Spain, 2005.
10. J. Garcia-Alfaro, M. A. Jaeger, G. Mühl, and J. Borrell. Decoupling Components of an Attack Prevention System using Publish/Subscribe. In *2005 IFIP International Conference on Intelligence in Communication Systems*, pages 87–98, Montréal, Canada, 2005.
11. J. Hochberg, K. Jackson, C. Stallins, J. F. McClary, D. DuBois, and J. Ford. NADIR: An automated system for detecting network intrusion and misuse. In *Computer and Security*, volume 12(3), pages 235–248. May 1993.
12. K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, 1995.
13. R. A. Kemmerer. NSTAT: A model-based real-time network intrusion detection system. Technical Report TRCS97-18, Reliable Software Group, Department of Computer Science, University of California Santa Barbara, 1997.
14. C. Kruegel and T. Toth. Distributed pattern detection for intrusion detection. In *Network and Distributed System Security Symposium Conference Proceedings: 2002*, 1775 Wiehle Ave., Suite 102, Reston, Virginia 20190, U.S.A., 2002. Internet Society.
15. R. Lippmann, J. Haines, D. Fried, J. Korba, and K. Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks*, (34):579–595, 2000.
16. T. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P. G. Neumann, and C. Jalali. IDES: A progress report. In *6th Annual Computer Security Applications Conference*, Tucson, AZ, USA, 1990.
17. A. C. Migus. IDMEF XML library version 0.7.3. <http://sourceforge.net/projects/libidmef/>, March 2004.
18. G. Mühl. *Large-Scale Content-Based Publish-Subscribe Systems*. PhD thesis, Technical University of Darmstadt, 2002.
19. P. A. Porras, and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *20th National Information Systems Security Conference*, pages 353–365, 1997.
20. D. Powell. Group communication. *Communications of the ACM*, 39(4):50–53, 1996.
21. M. Ruff. XmlBlaster: open source message oriented middleware. White paper [on-line]. <http://xmlblaster.org/>, 2000.
22. S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur. DIDS (distributed intrusion detection system) - motivation, architecture and an early prototype. In *Proceedings 14th National Security Conference*, pages 167–176, October, 1991.
23. E. H. Spafford, and D. Zamboni. Intrusion detection using autonomous agents. *Computer Networks*, 34(4):547–570, 2000.
24. S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS – a graph-based intrusion detection system for large networks. In *19th National Information Systems Security Conference*, 1996.
25. D. Veillard. The XML C library for Gnome (libxml). <http://www.xmlsoft.org>, 2006.
26. G. Vigna and R. A. Kemmerer. NetSTAT: A network-based intrusion detection system. *Journal of Computer Security*, 7(1):37–71, 1999.

27. G. B. White, E. A. Fisch, and U. W. Pooch. Cooperating security managers: A peer-based intrusion detection system. *IEEE Network*, 7:20–23, February 1999.